

CIRCULATION COPY
SUBJECT TO RECALL
IN TWO WEEKS

UCRL- 94377
PREPRINT

Interactive Wrapper Tool for Mapping Objects to Fortran Library Functions

B. S. Lawver
Computer Systems Research Group
Electronics Engineering Department

This paper was prepared for submittal to
ACM Conference on Object Oriented Programming
Systems, Languages, and Applications
Portland, Oregon
September 29-October 2, 1986

April 1, 1986

Lawrence
Livermore
National
Laboratory

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement recommendation, or favoring of the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Interactive Wrapper Tool for Mapping Objects to Fortran Library Functions*

*B. S. Lawver
Engineering Research Division
Lawrence Livermore National Laboratory
Po Box 808 L-156
Livermore, Calif. 94550
415-422-6234*

April 1, 1986

Abstract

Many of us would like to start the development of a new software system fresh with no constraints, but often we are given a few minor conditions such as here is a library of 150 Fortran functions that must be part of the new system. So step one in the analysis phase becomes, "do we have to use Fortran for the whole project or can we ignore the 150 library functions?" After all of the arguments have been heard and you have adopted the enlightened approach of using an object-oriented programming system to develop the new system, how do you proceed to *wrap* the 150 library functions? We looked at this problem of including a large group of library functions and chose to construct an interactive tool to build the data interfaces.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Introduction

The EAGLES project of the Electrical Engineering Department of Lawrence Livermore National Laboratory is investigating user interfaces¹ and data structures of computer-aided engineering tools and software environments² for constructing new engineering tools. Some of the first tools we developed included a new environment for our controls engineers. They have a large collection of verified Fortran functions that they use to solve design and simulation problems. In the past they have solved their problem by writing a new Fortran program that called the appropriate functions. The EAGLES project developed an interpreter that can construct their data elements and call the library functions. The interpreter was written using Objective-C³ an object-oriented programming language. After demonstrating the interpreter, we looked at an automated wrapper generator to construct the interface modules that would map the calling arguments from the interpreter function call to the Fortran function and map the return values back to interpreter objects.

Careful study of our set of Fortran functions indicated that for most functions there was a consistent set up, call and return. Since each interface function would be a procedure of several hundred lines and there was considerable similarity, we chose to investigate an automatic procedure generator that from simple input could output the ready to link interface module. This tool became known as a wrapper generator. The intended user of this tool is the software specialist in the controls group who is responsible for the catalog and library of Fortran functions. This person is already familiar with the call interface to the Fortran function but has little or no experience with object-oriented programming. The main objective of the tool is to build the interface from input that is normally found in the header or catalog for the library function.

Wrapper is a term that evolved within our work to characterize the interface between non-object-oriented software and software that must be used but exists in a form that is not object-oriented. At this time it is not meant to be a general solution for the data interface between all objects and any functional code but rather a specific solution for our set of applications. For some functional codes or applications, there are no sources available to allow the wrapper to be incorporated into the code. This case of the wrapper problem implies that the unmodified code is embedded in a black box with input and output feed to the box by a wrapper. We are not yet to the point of capturing every keystroke from our user interface and reformatting them for each

application or the opposite case of capturing all characters on output from the code and either inserting them into our object database or reformatting them for our user interface. Our term wrapper was first coined by Dr. Cox⁴ and is similar to Dr. Joy's guardians⁵.

The portion of the wrapper concept that we studied is the integration of existing code libraries with our new set of engineering tools or codes. Since the code libraries are specific to engineering codes and our tools are also, we have narrowed the mapping of data to a limited but interesting set of data conversions problems. Clearly mapping all possible data structures definable in higher level programming languages to objects is ambitious and beyond the scope of our project⁶.

Hand-coded Wrapper

Before building an automated wrapper generator, a template for the function wrapper must be designed. The body of the wrapper is shown in figure 1. The main features of the wrapper are creation of local work arrays, mapping of objects to Fortran parameters, the call of the function and the mapping of the return values from the Fortran function to a new object. Except for the call each of these sections can be described in terms of messages to objects. Having reduced the wrapper to a small set of objects and messages, it is reasonable to investigate an automated code generator to be used as a wrapper tool or generator.

The objects of interest to the wrapper are the arguments collected by the interpreter, local work arrays required by the Fortran function, and objects to return the result of the function. The result and local work array objects are normally global data structures that the Fortran program declares in order to use these functions. A nice side effect of using an object-oriented environment is being able to postpone the creation of storage for these arrays until actual execution of the wrapper. A single message to an argument can evaluate the expression that was passed, test for the legal data type that Fortran will expect, and make a copy of the data. The copy solves the problem created by Fortran which allows the nasty side effect of assigning to its calling arguments and our interpreter allows no side effects. Additional messages to the evaluated argument will reply the size of arrays which can be used either as arguments to the Fortran function or parameters to messages used to create the return and work array objects.

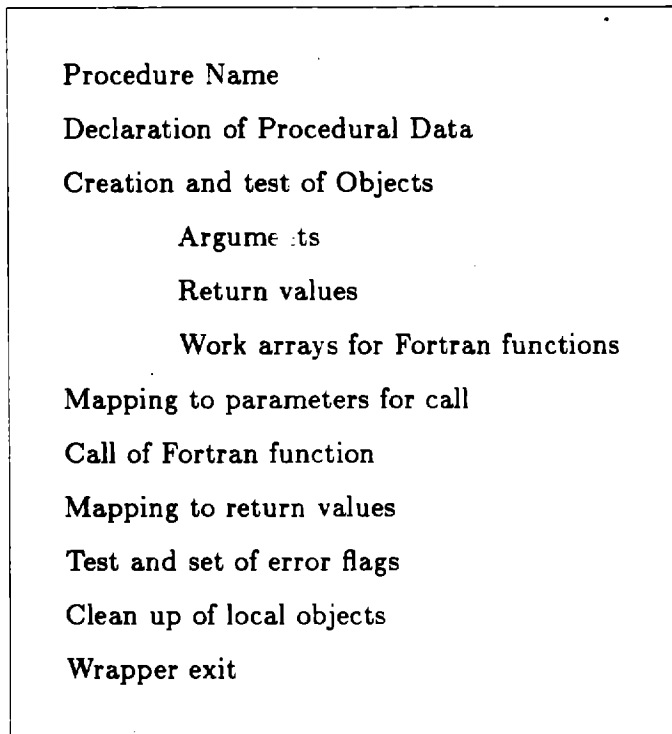


Figure 1. Outline of Wrapper Procedure.

The mapping of objects to Fortran functional arguments uses C pointer types and the Fortran standard of call by reference. Our data objects have methods which return pointers to their instance and indexed variables. As a concession to the problem that C and Fortran store arrays differently, EAGLES stores arrays as Fortran does to avoid having to transpose C arrays for each Fortran call. The methods required to handle all possible mappings between EAGLES objects and Fortran arguments is actually a very small set of methods. Our Fortran library functions treat complex double precision matrices as two double precision arrays of data. EAGLES matrix objects contain both the real and imaginary parts in the object with the real array stored first and the imaginary part stored next. It can return a pointer to either the real or imaginary part. Actually one EAGLES matrix contains the information necessary for mapping several Fortran library function arguments. Fortran allows for run time sizing of arrays in functions but requires one of the arguments to pass the true size. EAGLES matrix objects support methods to return their dimensions besides pointers to the array data.

The last part of the wrapper is a dummy method that at initialization binds the address of the wrapper into the call table for interpreter functions. C supports jump tables but Objective-C

objects can not easily save and restore addresses in a jump table. We could constantly update the jump table to include link time binding of pointers to new wrappers or simply design the wrapper so that if it exists its entry is added to the jump table. The later method will make updates easier and limits the references to wrappers to just what is in the class table for the interpreter.

Interactive Wrapper Tool

The interactive wrapper tool consists of five separate forms or worksheets which collect information about the data and function mappings, and a code generations module. As the user answers questions presented by each worksheet, objects are created to store the answers. Later when all questions are answered a single message to the collection of objects is sent which causes the code generator to sequence through all of the answers and write the code.

Add function

M function name: *world*

Description (≤ 40 chars): *opens the universe*

Number of arguments: *2* Arguments worksheet...

Number of returns: *2* Returns worksheet...

Name of external to wrap: *universe*

Fortran function or subroutine (f or s): *f*

Number of arguments in
calling sequence of external: *6* Calling sequence worksheet...

Number of error tests required: *1* Error test worksheet...

OK... cancel...

Figure 2. User interface for Wrapper Generator.

The first form of the user interface (figure 2) asks the user to supply general information about the function that is to be wrapped. Most of the information can be found in the catalog synopsis of the library routine. The wrapper user does have to decide what name the language interpreter will use and how many arguments are needed to call the function from the interpreter.

The EAGLES project chose the name EIG with one argument as the interpreter function that returns Eigenvalues and Eigenvectors. The library function which implements this particular function requires fourteen arguments to compute the EIG function on a single complex double precision square matrix.

Answers to the questions in the first form lead to additional worksheets which gather more information about the specific mapping between library function and interpreter function. Specifically the size and type of the return values and arguments are needed (figures 3 and 4). The arguments have their own size, type, and shape as part of their object, therefore this information is used to test and validate the data passed to the Fortran library function before Fortran is allowed to abort. The return values must be created and storage allocated so that Fortran can write the output into appropriate locations. The size of each return value can be determined by a function of the input arguments. For example, the return might be a row or column slice out of an input argument or has a row size equal to one argument and column size of another. External work arrays for the Fortran functions are declared by a form similar to the returns worksheet. These work arrays are used by Fortran library functions to keep the library function small but useful for large problems and force the allocation of space to the user of the library function.

Argument Worksheet

M function:

world(A1,A2)

Argument	Type
A1	<i>square matrix</i>
A2	<i>vector</i>

OK... cancel...

Figure 3. Worksheet for data typing interpreter arguments.

The worksheet titled *Calling Sequence Worksheet* (figure 5) collects the information needed to build the mapping from object to Fortran argument. Here mappings such as *real part of*, *imaginary part of*, *row size of*, or *column size of* describe the relationship between each Fortran

Returns Worksheet			
M function { R1,R2} = <i>world</i> (A1,A2)			
Return	Type	Size	
R1	<i>matrix</i>	<i>row(A1)</i>	<i>column(A2)</i>
R2	<i>matrix</i>	<i>1</i>	<i>column(A1)</i>
OK...		cancel...	

Figure 4. Worksheet for mapping return values to function elements.

function argument and objects such as *argument 1*, *return 2*, or *work array 5*. Since each Fortran argument can be both an input value and output value, a separate mapping is required for each Fortran argument. Input only arguments will have *none* as an output mapping and output only arguments have *none* for its input mapping.

Calling Sequence Worksheet		
external FUNCTION <i>universe</i> (X1,X2,X3,X4,X5,X6)		
M function { R1,R2} = <i>world</i> (A1,A2)		
Arg.	Input Data Linkage	Output Data Linkage
	operator (operand)	operator (operand)
X1	<i>real (A1)</i>	<i>none</i>
X2	<i>row (A1)</i>	<i>row (R1)</i>
X3	<i>real (A2)</i>	<i>real (R1)</i>
X4	<i>imag (A2)</i>	<i>imag (R1)</i>
X5	<i>none</i>	<i>real (R2)</i>
X6	<i>none</i>	<i>none</i>
OK...	cancel...	

Figure 5. Worksheet to define mappings for function elements.

Certain Fortran arguments are used for error flags and have a special meaning that should be sent to output. This mapping of Fortran function arguments is defined by use of another form. This worksheet asks the function argument, a test operator, and threshold value. If the condition is met then the error message is reported to the language interpreter error processing function. For now, *less than*, *greater than*, and *equal to* are the available tests for integer values. This worksheet is shown in figure 6.

Error tests on function return values			
Argument	Test	Value	message
X6	>	0	This is terrible, I think but don't quote me, please.
OK...	cancel...		

Figure 6. Worksheet for error handling.

Progress

The history of this project is interesting. The scope and requirements for this tool were laid out with a proposed user interface. The tool was then shelved for about 6 months while development of the interpreter proceeded. When the *function call* grammar was implemented in the interpreter, then the template design of the wrapper was finished. A coded generator was implemented and was able to duplicate the template. Since much of the work of the template wrapper was implemented as messages to objects of the interpreter both the code generator and the wrapper were simple and powerful. The wrapper was tested and verified with an Eigenvalue/Eigenvector library routine.

Again the wrapper generator was shelved for another 6 months while the user interface⁷ tools were developed. During this time more development on the interpreter proceeded so that some of the code necessary was simplified. Two very interesting bugs though showed up during this time. The interpreter was designed to store data in a column major form to be compatible with the Fortran library functions and avoid a transpose copy on input and output from library routines, but the implementer managed to code row major storage. The Eigenvalues for the test matrix and its transpose were identical so this bug went undetected for some time. The other bug occurred

because the wrapper implementer could not interpret the catalog documentation for the Eigenvalue function and passed the column size for the packing factor. Again because it was a square matrix the packing factor happened to equal the column size. This bug demonstrates the problem with wrapping Fortran library functions to objects, where one expert understands objects and another understands library functions. Somehow one of the expert's knowledge must be captured and built into a tool.

The user interface was completed by a user of Fortran library routines. This captured the user knowledge and demonstrated the power of our interface tools. While implementing the user interface for the wrapper tool, the user became experienced with the object-oriented tools we were using and he found he could build wrappers without the tool. This supports the original point that the mappings are simple enough to be implemented with a small set of messages to the various objects and once you understand messaging, then building wrappers is simple.

Conclusion

When we charted this project, we anticipated that the tool might be as complicated as the wrapper itself. In fact, we considered a well annotated template from which to build wrappers. The training and background required to use and understand a template was judged to be too large of a barrier for intended user. We expect that the tool will be used both as a training device to examine the mapping from objects to Fortran library routines and for at least the initial wrappers. Later the user may decide to use a template either to code simple functions that he doesn't have a library function for and the language doesn't support. Also, the user may be forced to hand code parts of the wrapper because the mapping is more complicated than our tool supports. In either case, having the wrapper tool has made the users life more productive.

References

1. James D. Foley, Victor L. Wallace, and Peggy Chan, *The Human Factors of Computer Graphics Interaction Techniques*, *IEEE Computer Graphics and Applications*, IEEE Computer Society Press, November, 1984, pp. 13-48.

2. Grady Booch, *Object-Oriented Development*, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, IEEE Computer Society Press, February, 1986, pp. 211-221.
3. —, *Objective-C Reference Manual*, Productivity Products International, 1984.
4. Brad J. Cox, personal communication, Nov. 1984.
5. Kenneth I. Joy, *A Model for Graphics Interface Tool Development*, *Graphics Interface '85*.
6. Sandra A. Mamrak, Michael J. Kaelbling, Charles K. Nicholas, *An Approach to the Solution of Data Conversion Problems in Heterogenous Networks*, submitted to *Transactions on Computer Systems*, Aug. 1983.
7. Margret E. Poggio, *Developing a Portable Window/Menu Interface Using Object-Oriented Programming Tools*, submitted to *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, April 1986.